

## Maintenance Panel

The maintenance panel (MP) is a subsystem designed to allow observation of the system and manual programming of devices. It can both read data from the bus onto display lights and write it from a set of switch settings, in either bus master or slave modes.

In master mode, the maintenance panel must establish itself as bus master before it can perform any I/O to the bus. This protocol is implemented directly in combinational logic. When the bus-master logic sees MP-BR' asserted it knows the MP wants to become bus master and asserts BR' (bus request). It then waits for BG-OUT (Bus Grant out) to go low, indicating that no other device is waiting to become bus master. When this falls it waits for BG-IN from the priority arbiter to signal that it is next in line to use the bus. When BG-IN comes high it signals the state machine that it has control of the bus (GOT-BUS) and sets a flip flop so that its bus grant is not passed to any other modules. The state machine then waits for BBSY' to become de-asserted, indicating the previous master was finished with the bus and it can use it. This signal is passed to the state machine as BUS-FREE. The MP then asserts BBSY' until it has completed use of the bus. When the I/O is completed (indicated by MP-BBSY dropping) the flip-flop is reset, BBSY' is de-asserted and the Bus grant line is no longer blocked.

The general sequencing of both the bus master protocol and the slave protocol is done with a sequencing state machine, as follows:

```
000 [Idle]
   | <- GO signal from MP
   |
001 [Generate MP-BR']
   | <- Got-Bus
   |
010
   | <- Bus-Free
   |
011 [Assert BBSY']
   |
100 [Assert MP-DRIVE and MSYN']
   | <- wait for SSYN from slave
   |
101 [Assert BBSY', drop MSYN']
   | <- wait for SSYN' from slave to de-assert
   |
110 [De-assert BBSY', load counter to state 000 (or 001)]
```

## Maintenance Panel (con't)

The state machine is implemented using a counter, selector and decoder. The counter enable is connected to the output of the selector, and the input lines of the selector are tied to the corresponding control lines to allow the counter to advance when the line for that state turns high. The state variables are then decoded and fed separately to the rest of the system.

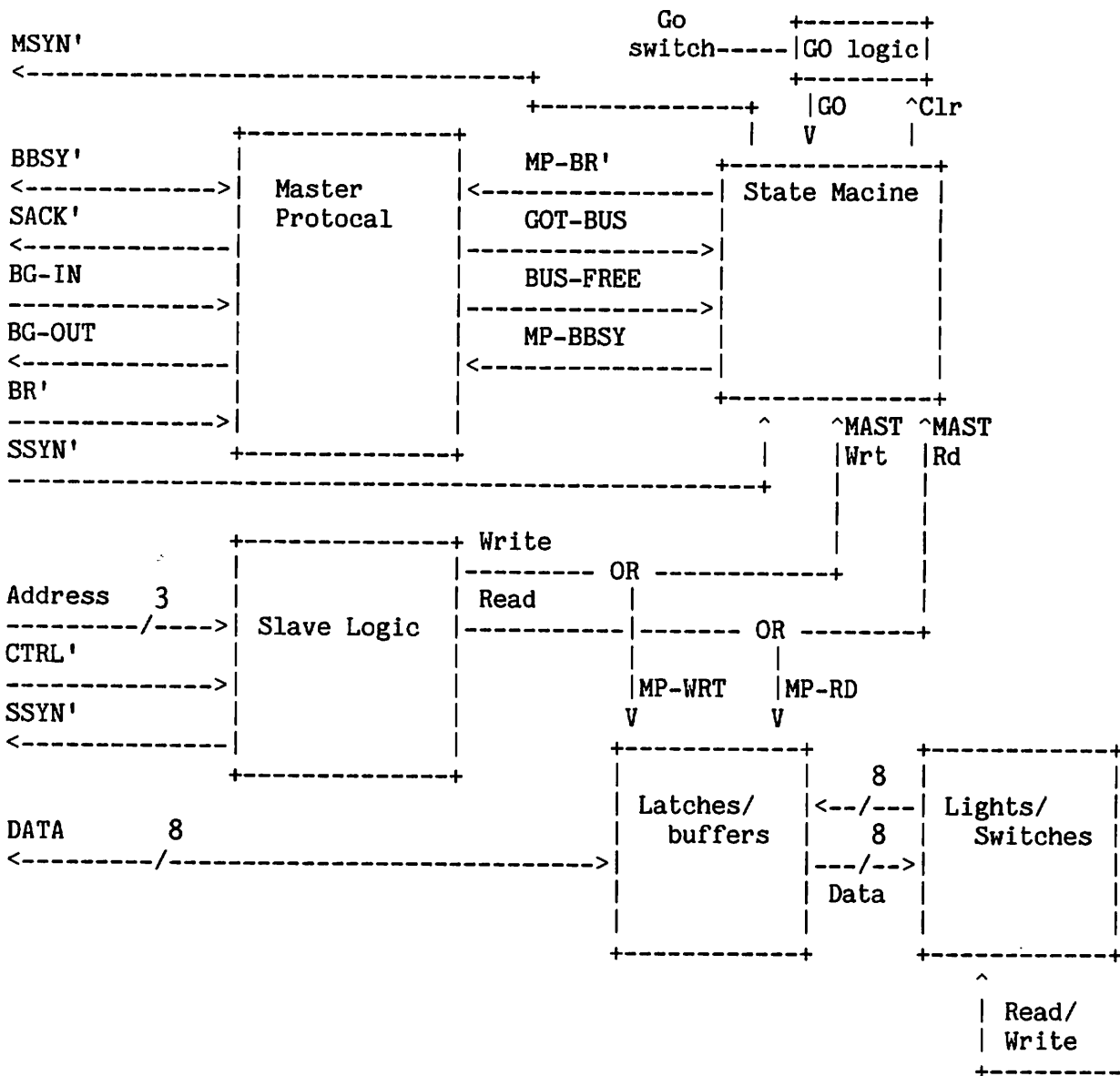
The initial GO signal is provided by first debouncing a switch and feeding the output of this to the CLK input of a flip flop. When the GO switch is activated, the leading edge causes the flip flop to toggle, allowing the counter to proceed from state 000. The flip-flop is reset when the counter is loaded. An addition was made to the circuit allowing the counter to be loaded to state 000 as well as 001 depending on a switch setting. This causes the MP to execute another bus cycle immediately instead of stopping and waiting for GO. This is VERY handy for finding timing problems, since the repeating signals can be easily traced with a scope.

The MP controls the address bus by a series of switches gated to the address bus. These are driven on to the bus when MP-DRIVE is asserted. If the MP is in Write mode, the MP-DRIVE signal will cause MAST-WRT' to be asserted along with MP-DRIVE, allowing the data in the switches to be driven onto the bus. If the MP is in Read mode, MAST-RD is asserted and the data on the bus is clocked into a latch with outputs driving a set of LED's.

The MP can also act as a bus slave, responding to address 101xxxxxxx. When the address is selected and MSYN' is asserted, either a read or write signal is generated. These are ORed with the MAST-RD and MAST-WRT signals generated in master mode to drive the MP-RD and MP-WRT signals controlling the latches and data buffers. SSYN' is generated along with the data action.

I realize that the maintenance panel operation is pretty straightforward, but your description (the last two paragraphs) is difficult to follow. When indicating the cause-effect relationships of signal assertions, it really helps a great deal to have an accompanying diagram.

# Maintenance Panel Block Diagram



## Maintenance Panel Signals

Got-Bus	Indicates MP is next in line to be Bus Master
Bus-Free	Indicates MP is free to assert $\overline{\text{BBSY}}$
$\overline{\text{MP-BR}}$	Asserted when MP is ready to start master cycle
MP-DRIVE	Asserted when MP is actually driving the bus ( $\overline{\text{MSYN}}$ cycle)
MP-RD	Clocks data into lights register
$\overline{\text{MP-WRT}}$	Drives switch data onto bus
MAST-RD	True when Master-mode MP wants to write to the bus
$\overline{\text{MAST-WRT}}$	" " " " " " " " read the bus

*↑  
good way to indicate level of assertion (1, 0)*

## Maintenance Panel Design and Construction notes

A feature was added to the MP to allow it to work in 'continuous' mode, as described earlier. This was very useful in not only debugging the maintenance panel, but the peripherals as well, since a repeating waveform is generated that can be observed with a scope.

A major construction annoyance was in the timing of writing data onto the bus. The problem stems from the fact that the Slave mode for the MP was added as an afterthought. In the original design, MAST-WRT' drove MP-WRT' directly, so there was only one gate delay between MP-DRIVE' and MP-WRT'. (which made it about even with MSYN', since it also has one gate delay from MP-DRIVE'). But when the slave mode was added, extra layers of logic were needed between MP-DRIVE' and MP-WRT' to allow for the case when the MP wrote to the bus as a slave. It turns out this delay was too long, since when a slave (e.g., the MP) saw MSYN' asserted the data wasn't ready yet. As a kludge to get around this, an extra switch was added so that when the MP is in WRITE mode as a master, MAST-WRT' and MP-WRT' are connected directly together, but in READ mode-as-master it can respond as a slave. (the setting of the switch should be in READ mode for operation as a slave that can write. It can properly read when in this position.)

System clock

BBSY

MASTER SACK

BR

B4I

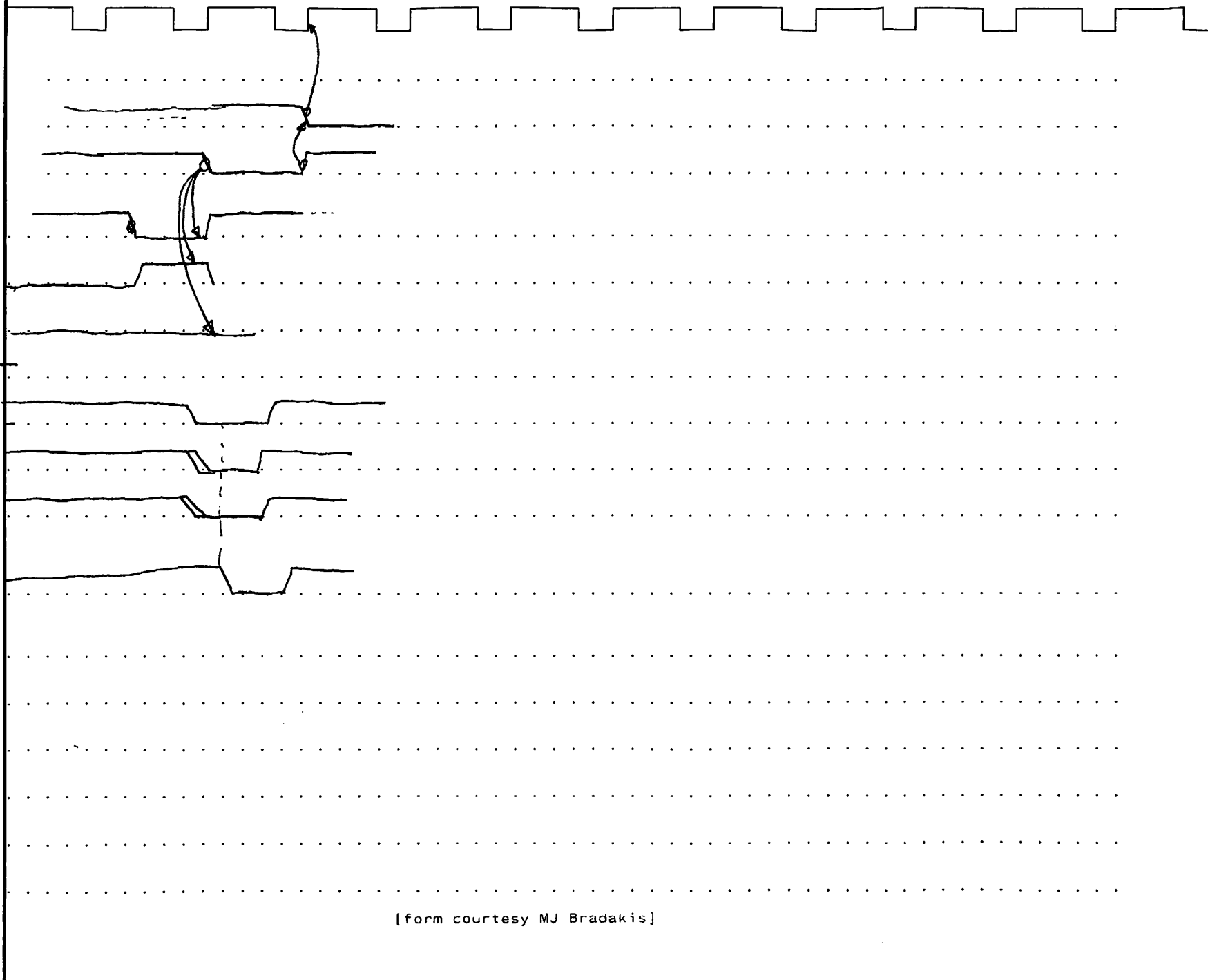
B4O

$\overline{MSYN}$

SLAVE ADR

DAT

$\overline{SSYN}$



## Lab 1. System Clock and UART device

The system clock provides the timing for the rest of the system. It is based on a crystal oscillator running at 1.8432 Mhz. This signal is then fed into into the clock circuit to produce the 618.04 Khz non-symmetric signal for the rest of the machine.

### System clock design

The output from the crystal oscillator is first fed into an inverter, then is fed into a 74C160 counter. The counter's inputs are all grounded, so the load input to the counter is effectively the same as a synchronis clear.

When the clock is running normally, the RUN line will be high, turning on the ENABLE lines on and allowing the counter to count. When RUN goes off, the counter is disabled and the clock output stops.

Bit 2 of the counter ( $2^1$ ) is tied through an inverter to the LOAD input of the counter (active low). This causes the counter to be reset on the next clock edge after reaching 10. Since the load input now goes low ever third clock pulse, this provides the desired system clock (see the timing diagrams for more info).

The counter can also be stepped a single pulse by using the single step switch. The switch is first debounced through an RS flip-flop, then fed to the clock of a 74C74 D type flop flop. A rising edge on the CLK turns on the Q output of the flip-flop, which in turn enables the counter. When the counter reaches 10, a low is asserted to the flip-flop's CLR input, bringing the Q input back to zero, and disabling the counter.

The operation of the UART is described, but much of the other important documentation is not here. Most important are the block diagrams indicating control + data flow, and a glossary of the signals for each part.

Also, as long as you will be using this format for your book, I'm afraid you'll have to organize it better (splurge on some dividers). Separate your write up from the handouts and other notes.

5/10

Bob

## The UART

### Bus interface.

To prevent loading and fan-out problems, all lines from the buss are buffered. In this case there are only two of them SYSCLOCK' and INIT'. Since INIT' is generated by a switch (and therefore immune to fan-out problems) I see no sense in buffering it. The SYSCLOCK' input is first fed into an inverter that generates EARLY-CLOCK. This in turn is fed to another inverter producing M-CLK. EARLY-CLOCK is used in instances where a logic element must be clocked through another gate. Since EARLY-CLOCK is both inverted and occurs one gate delay sooner than M-CLK, the clock skews caused by clocking the logic through the extra gate are hopefully minimized. M-CLK is fed into a divide by 16 counter to produce an 8\*4800 baud signal called BAUD-CLK. This signal is first NANDed with EARLY-CLK to help compensate for delays induced by the counter (Hayes advice). An EARLY-BAUD-CLK is provided for much the same reasons as EARLY-CLK was for system use; except EARLY-BAUD-CLK is used for the baud rate timing elements.

### The Receiver

The receiver is based on a simple state machine implemented with a counter. when the counter is idle (assuming initialized by INIT'), the C74 flip flop is reset, so the baud-divider counter is disabled. The state machine counter is driven the baud-divider, which divides the 8\*4800 baud BAUD-CLK down to a frequency for sampling the incoming signal. When the machine is at state 0000, the  $2^4$  output of the divider loads (i.e., synchronisly resets) the counter. On the other states, bit 8 performs this function. The reason for this is to center the clock edge of the shift register in the the middle of the incoming data bits (See the timing diagram).

When a start bit comes on the serial data in, the flip-flop is set, enabling the counter. After the first four pulses, the shift register is clocked and the counter is incremented. The states are used as follows:

0000	idle, counter loaded
0001	start bit
0010	LSB
:	
:	
1001	MSB, Recv.REQ pulled high.

When state 1001 is reached the flip-flop is cleared and counting stops, and Recv.REQ is pulled high. The state machine remains in state 1001 with RECV.REQ high until RECV.ACK turns on. This clears the counter, causing RECV.REQ to drop.

## The Transmitter.

The Transmitter works in much the same fashion as the receiver, using a counter as the basic state machine. When it is in the idle state (or initialized) the counter is at state 0000, and the flip-flop is reset, turning the baud-divider counter off. When XMIT.REQ is received, the flip-flop is set causing the counter to be enabled. This causes the counter to go through the following states:

0000	idle, loads shift reg
0001	shift out stop bit
0010	shift out start bit
0011	LSB
:	
:	
1010	MSB, counter disabled, XMIT.ACK asserted

When state 1010 is reached and XMIT.ACK is asserted, the counter stays in 1010 until XMIT.REQ drops. This action clears the counter and resets the machine.

Note the Start and stop bits are encoded by feeding lines directly into the shift registers. The convention STOP-START-<data>-STOP is used instead of START-<data>-STOP-STOP because the timing of the first bit is dependant on the state the counter was last in. Because the timing on START bits is much more critical than the STOP bits, the extra STOP bit is sent first.

Note on both the receive and transmit state machines the clock is provided by combining the BAUD, EARLY-CLK and EARLY-BAUD-CLKs, in hopes of keeping the response of the ACK and REQ lines in synch with the rest of the system.

### Construction notes:

The circuit for the UART was densely packed on the board and required 16 packages to implement. It was initially tested with the single step switch, then further tested by hooking the serial output of the XMT module to the RECV module. By setting the data with switches and watching the output with lights and switches, the operation could be observed with the free running clock.

The major design blunder encounter was wiring the receiver state machine so it initialized in a hung state. (State 0 loaded the counter responsible for incrementing the state machine to state 1, but since it was loaded to zero, it couldn't count, so....). This was repaired by altering the design of the receiver baud-rate timer so bits 4 and 8 were used to provide the sampling timing, instead of using the load inputs. Unfortunately, this modification added an extra layer of gates to the state machine clock circuitry, but this still appears to be (and hopefully, IS) well within the timing constraints of the rest of the machine.

## UART Signal definitions

Serial Data in- Data from terminal (positive logic)

Baud Clk- 8\*4800 baud timing

Recv.ACK- Signal from term interface indicating data was taken. Should be high until Recv.REQ drops

Recv.REQ- Indicates character available. Remains high until Recv.ACK is asserted

Serial Data out-Data to terminal (negative logic)

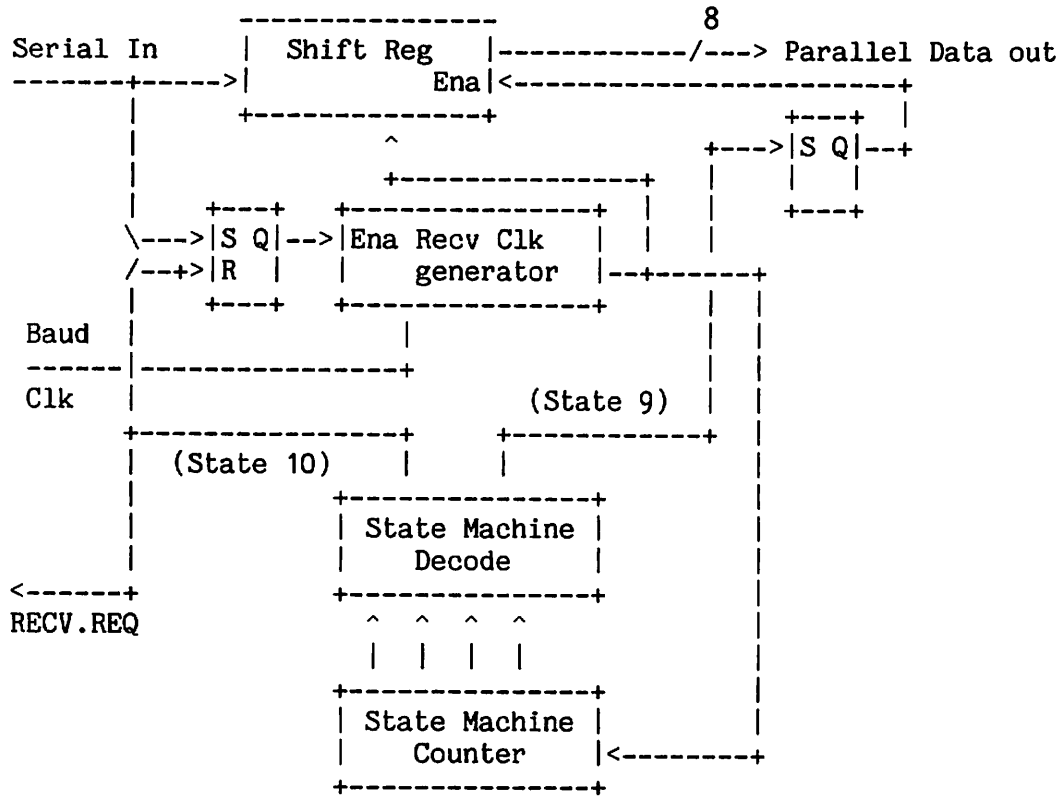
Xmit.REQ- Requests that the character be sent. Should remain high until Xmit.ACK

Xmit.ACK- Asserted when data taken, falls when transmit is complete.

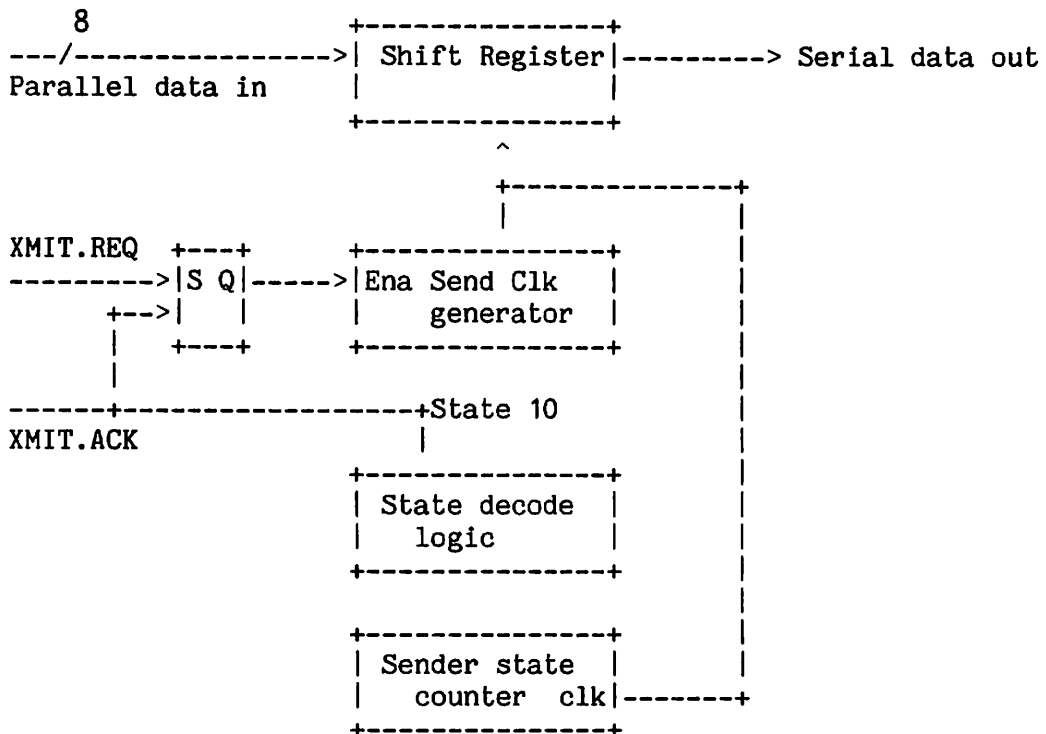
Early Baud Clk- Inverted 8\*4800 baud signal, occurs 1 gate delay sooner.

# UART Block Diagram

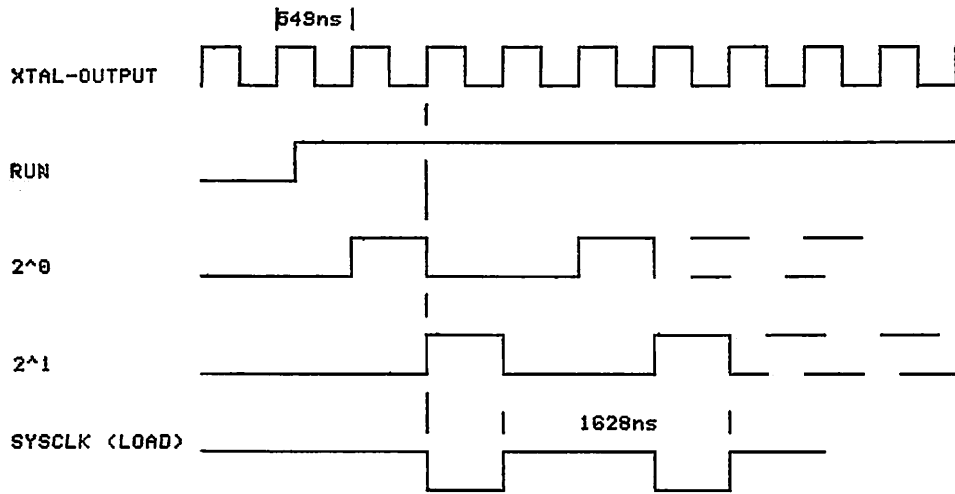
## Receiver-



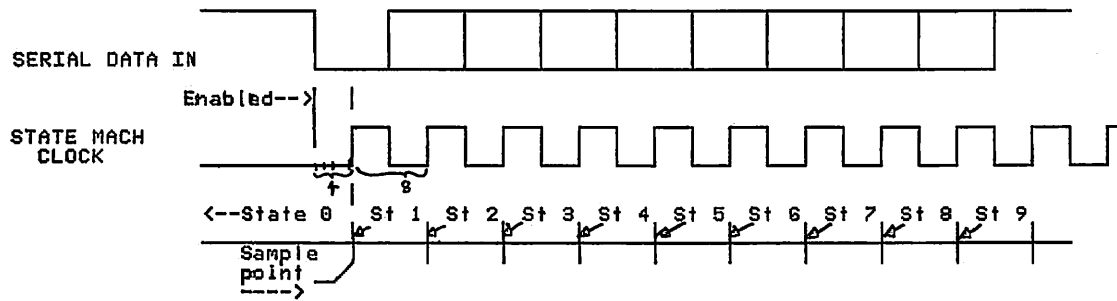
## Sender -



### System Clock Timing



### UART (Recv) Timing



## Lab 2- Terminal interface.

The terminal interface provides the necessary protocols for the UART to talk to the system bus, and provides the necessary buffering for the input and output data lines.

The selection portion of the Terminal interface detects if the address of the module is selected. It only checks the top three bits, and the low order bit for the pattern 100xxxxxxxL. If the low order bit is a zero and the bus is being read, the status register is read to the bus. If the low order bit is a one, then the UART is selected.

The status register simply drives the RECV.REQ and XMIT.ACK lines (Data lines 0 and 1, respectively) onto the bus, and immediately asserts SSYN'. On the next bus cycle, the status is read, and MSYN' drops, causing the data and SSYN' to become de-asserted. This is used for the processor to poll the status of a uart transmit or receive.

To write to the UART, the bus master generates 100xxxxxxx1 on the address bus and ~~does not assert~~ CTRL'. This generates the signal TERM-WRITE, causing a flip flop to set that in turn asserts XMIT.REQ. The data will be read from the bus into the shift register. Since this takes several clock pulses (because the shift register must wait for the slower 8\*4800 baud-clk to load) SSYN' is not asserted until XMIT.ACK comes high, indicating the data was taken. When XMIT.ACK is asserted SSYN' is allowed to go low, and the data can be taken off the bus. However, the flip flop controlling XMIT.REQ cannot reset until XMIT.ACK becomes low again. This prevents the bus master from attempting to send another character until the first one is sent, and the bus hangs until the first transmission is complete.

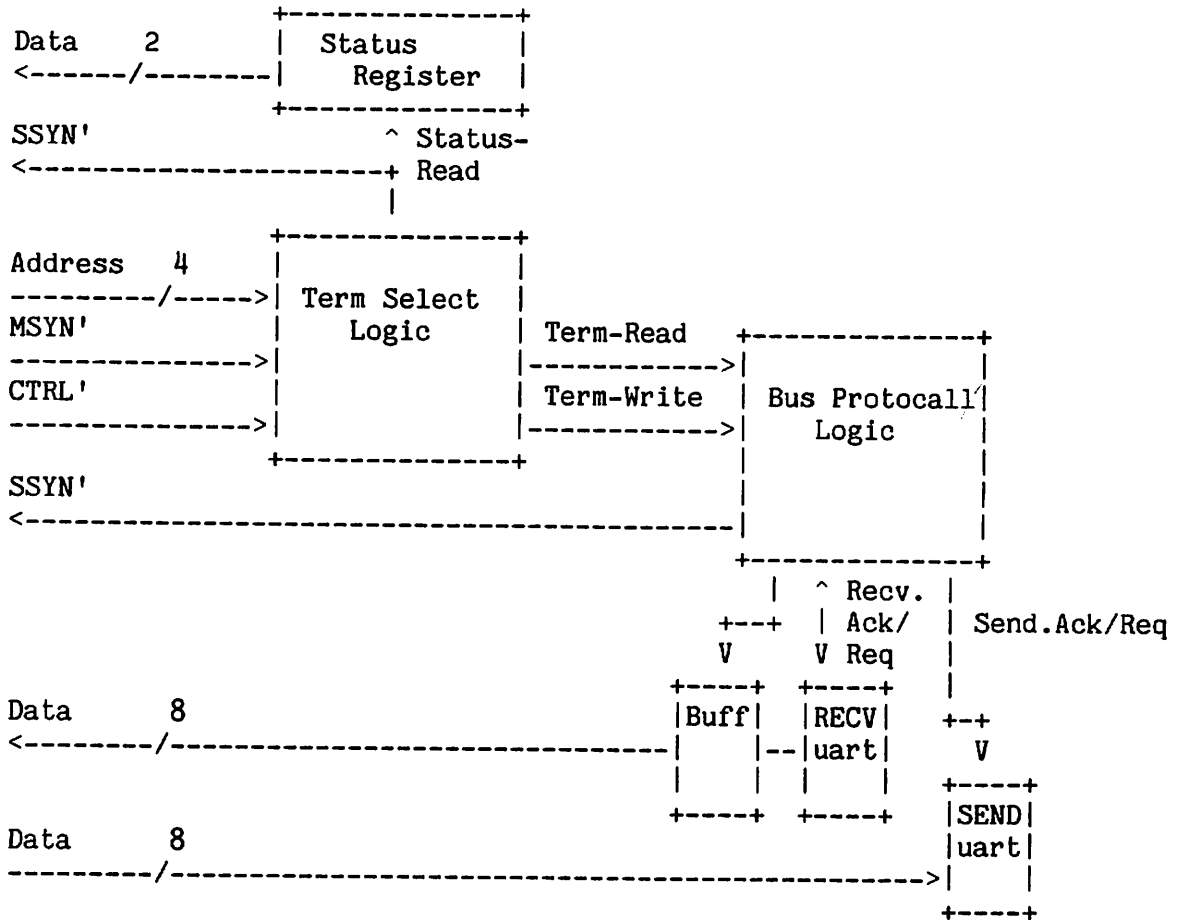
To read the uart, the bus master generates the address 100xxxxxxx1 and does not assert CTRL'. If a character has been received, RECV.REQ will be high and the flip flop sets, driving the data to the bus and asserting SSYN'. This also generates RECV.ACK, causing RECV.REQ to fall. After RECV.REQ falls and TERM-WRITE falls (MSYN' no longer asserted) the flip-flop resets and the data is taken off the bus, and SSYN' is de-asserted.

## Terminal Interface Signal Dictionary

(see also UART definitions)

- Term-Read:** High when the term interface's address is on the bus and MSYN' is asserted, and CTRL' is not asserted. Indicates character is to be received.
- Term-write:** High when the term interface's address is on the bus and MSYN' is asserted, and CTRL' is asserted. Indicates character is to be transmitted
- Status-read:** High when the terminal status register is selected, MSYN' is asserted and CTRL' is asserted.

## Terminal Control Block Diagram



## Lab N+1- CPU

Mainly due to the fact that I actually got it working, this isn't going to be a heavy duty description of how the CPU was designed. It is just a series of short notes on oddities that were specific to my design.

The register layout of the CPU followed the layout given in the handout almost exactly. Tri-state 374 latches were used for most of the registers, except the Accumulator and the condition codes. The condition codes were implemented with a four bit latch and then buffered with a 244, so the carry bit could be made available to the ALU without also enabling it to the micro bus. The condition codes were duplicated on both the hi and low order address lines, so they would be in the right place for both the shift instructions and also the conditional branching (more on this later).

The accumulator was implemented with two 74C195 registers. The ability to shift right was created by hooking them up backwards (Unfortunately I didn't realize this until AFTER I built it...). Thus the hi order data went into the low order bit of the register. The accumulator formed the B leg of the ALU and the "Micro Bus" (output of all of the other registers, except the address regs). went to the A leg of the ALU. The registers all read in data from the "Destination Bus" (Dest Bus in the schematics).

Except for the accumulator, all registers were selected via 74C42 decoders. The multiplexer for the data bus was also controlled by the same decoder as the register enables. This caused several bugs in the microcode, since it wasn't immediately obvious that to pass data from the accumulator to the Dest bus required one of the registers to be enabled, (even though it's contents was ignored) otherwise random data from the bus got gated to the registers.

The Microcontrol. This was somewhat different than the one in the handout. First off, no hold register- the microcode was just too simple to need it. Second, there was no mux for selecting the micro branch address. I got around this by ORing the top four bits of the MicroBus to the bottom four bits of the LIT field input to the counters. If I wanted to branch to the Lit field with no conditional, I simply didn't enable any of the registers (pull down resistors assured a zero value). To perform an instruction conditional jmp, I just enabled the IR-H, and for condition code jmps the CC register was enabled. This is why the CC bits were placed on both the upper and lower four bits.

The bus control logic was stolen straight from the Maintenance panel (most of the chips were even in the same locations).

The Microcode, written with MakeCode (see manual) is pretty straight forward. It is very inefficient, but since we weren't graded on CPU speed I wasn't worried about that (many of the instructions could be folded, but weren't for clarity). About the only really arcane thing is the use of Anderson's methods for conditional branches on the condition codes and the STC and CLC instructions. (See Anderson's notebook for details.)

## BUGS/Conclusions

There were dozens of microcode bugs. I was pretty inept at that.

The accumulator was wired backwards, so it shifted left instead of right. (Jim G. pointed this out to me).

The 906 buffer and pull up were left of the carry input to the ALU. This caused the ALU outputs to go meta-stable on some occasions, but not others. e.g., it would perform an ADD instruction correctly, but it wasn't able to increment the PC from 07 to 08. Very wierd...it didn't show up until I tried to write a program longer than 8 instructions! (Thanks to John Slator and his logic probe for pointing this out.)

There was a wiring error (and another possible short) on the Micro Bus, causing register to load with strange values (thank god for logic analyzers).

There was a glitch/hazard in the logic selecting the UART SENDER causing it start without ever getting at acknowledged. (Hayes figured this one out).

The Bus control logic was re-designed to so it stopped the micro-pc on the u-Instruction that requested the bus, instead of the next one.

BUT, after the above (and many others) were fixed, it did correctly execute the test program. After five iterations of the test program (over a period of about two minutes) the CPU was powered off and torn apart. It probably has the record for the shortest operational life of any computer system in history.

## CPU signal definitions

Micro-control output lines:

LIT-0..LIT-10 These were the "Literal field" outputs of the UCS (micro Control Store). Mostly used for branch address (though in Bradakis's design they could also be written to the bus).

HR-CTL Used to control the non-existent Subroutine Hold Register.

LIT-CTL Loaded the LIT field into Micro PC.

LIT-ONLY Specified that the LIT field was to be gated into Micro PC without ORing in the MicroBus. This worked by disabling the MicroBus select decoder.

BUS-RW If high meant the CPU was reading the bus, if low it was writing.

BUS-REQ Meant the CPU was doing a main system bus read or write.

DEST-SEL-0..2 These were decoded to select which register was loaded from the dest bus.

PS-SEL If High, the PS was loaded from the Dest BUS, if low it was loaded from the ALU condition codes.

LD-PS Loaded the PS register.

LD-ACC Loaded (or shifted) the accumulator.

ACC-SHIFT If high, the accumulator was shifted, otherwise it was loaded.

A-SRC0..2 These selected which register would be enabled the the Micro Bus (A-leg of the ALU and input to LIT field).

CIN-SELO..1 Selected the carry bit into the ALU action 00= load with zero, 01= load one, 10 Load with PS carry, 11= Load with PS Carry inverted

ALU-CTOLO..3  
ALU-MODE Selects the ALU function

Other lines:

BUS-WAIT Stops the MicroPC from loading or counting while the CPU is waiting for bus ownership.

CPU-DRIVE Indicates the CPU is driving the main bus.

BUS-WRT Indicates the CPU is writing to the main bus.

DEST-DISABLE Prevents any of the Destination registers from loading (used during bus cycles).

ALU-CARRY Carry out of ALU

ALU-SIGN Sign bit of ALU output.

LD-ARH  
LD-ARL  
LD-IRH  
LD-IRL  
LD-PCH  
LD-PCL  
LD-T

Decoded "DEST-SEL" lines

IRH-ENA  
IRL-ENA  
PCH-ENA  
PCL-ENA  
T-ENA  
PS-ENA  
BUS-ENA

Decoded "ASRC" lines

Multiplexed main data bus onto the DEST bus.

CFU etc.

Well... we both know you can write good documentation (just from the software examples) — and we both know that you decided not to do a lot for the last part of the project — that's not at issue here.

I'd be way out of line if I were to ignore the contributions you made to the class and to the style of write-ups you have provided (i.e. the software tools you developed)

Certainly, if I were to evaluate the write-up for the last lab alone, it would come out short (no surprise!) but I ~~do~~ choose to add in the considerations mentioned above to this last lab. (You already have an A in the course, but I still feel the points should be made)

9/8/10

URTERM, URMP, & Bus

Generally, good documentation. Comments on specific items are made on the respective pages. I like the logic diagrams, but sometimes the arrows point the wrong way - this is not good. Your timing diagram sheets are nice - perhaps we could see if it's a popular idea and make them available for general use. (disregard, I just notice it's from Bradakis)

	Design	Hardware	Doc	
1	9	10	7 (revised)	26/30
2	9	10	9	28/30

HARDWARE

URUART 10

URBUS 10

URMP 10

URSTORE 10

MICRO 10

CPU - A00 1 to char 10 GBH.

CPU FINAL - Works! GBH 10

Documentation:

L1 7

L2 9

L3 9.5

Does All char's 2/Jan  
Relev